Docket: 34815/US (Formerly SPRODQ1100)

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

First Named Inventor:	Peter M. Mansour		
Appln. No.:	09/783,660		
Filing Date:	February 14, 2001	Examiner:	C. Zhong
Title:	Platform-Independent Distributed User Interface System Architecture	Group Art Unit:	2152

DECLARATION OF PRIOR INVENTION UNDER 37 CFR § 1.131

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

I hereby certify that this document is being sent via First Class U.S. mail addressed to Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450 on this 11 day of November, 2005.

Francesco Grobe
(Signature)

PURPOSE OF DECLARATION

1. This declaration is to establish completion of the invention of this application in

☒ the United States

☐ the NAFTA country _____ (name of country)

☐ the WIPO country _____ (name of country)

at a date prior to January 30, 2001, that is the effective date of the prior art

☒ publication

☐ patent

☐ patent publication

☐ other

that was cited by the

☒ examiner.

☐ applicant.

2. The person making this declaration is (are):

☒ the inventor(s).

☐ only some of the joint inventor(s) (and a suitable excuse is attached for failure of the omitted joint inventor(s) to sign)

☐ the party in interest (and a suitable explanation as why it is not possible to produce the declaration of the inventor(s) is attached)

FACTS AND DOCUMENTARY EVIDENCE

3. To establish the date of completion of the invention of this application, the following attached documents and/or models are submitted as evidence:

(check all applicable items below)

- ☐ sketches
- ☐ blueprints
- ☐ photographs
- ☐ reproduction(s) of notebook entries
- ☐ model
- ☐ supporting statement(s) by witness(es) (where verbal disclosures are the evidence relied upon)
- ☐ interference testimony
- ☒ disclosure documents

4. From these documents and/or models, it can be seen that the invention in this application was made:

- ☐ on _____.
- ☒ at least prior to January, 2001, which is earlier than the effective date of the reference.

DILIGENCE

5. The person making this declaration declare(s) that there was either reduction to practice prior to the effective date of the reference or conception of the invention prior to the effective date of the reference coupled with due diligence from prior to said date to a subsequent:

- ☐ actual reduction to practice.
- ☒ filing of this application.

TIME OF PRESENTATION OF THE DECLARATION

(complete (a), (b) or (c))

(a) ☒ This declaration is submitted prior to final rejection.

(b) ☐ This declaration is submitted with the first response after final rejection, and is for the purpose of overcoming a new ground of rejection or requirement made in the final rejection.

(c) ☐ This declaration is submitted after final rejection. A showing under 37 C.F.R. § 1.116(b) is submitted herewith.

DECLARATION

6. As a person signing below:

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on in-formation and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code , and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

SIGNATURE(S)

7.

Inventor(s)

Full name of **sole** or **first** inventor Peter M. Mansour
Inventor's signature [signature]
Date [ ] Country of Citizenship U.S.A.
Residence 696 16th Avenue West
Kirkland, WA 98033
Post Office Address (same)

Full name of **second** joint inventor Chad Arthur Schwitters
Inventor's signature [signature]
Date 10/27/05 Country of Citizenship U.S.A.
Residence 17615 NE 34th Ct.
Redmond, WA 98052
Post Office Address (same)

From: "Chad Schwitters"
Sent: Wednesday, December 27, 2000 12:37 AM
To: 'Takahashi, Mark'
Subject: RE: follow up discussion

[redacted]

3. operation of the client in a disconnected mode and "updating" the server after re-establishing a connection.

For disconnected mode, most of the client actually operates very much as it does in connected mode. The biggest difference is that it can't show any data it doesn't have already cached. For example, while disconnected, the user could drop the app menu (since it's in a permanent piece of cache) and select the mail application. Assuming the mail app has been run before and memory hasn't gotten desperately tight, the default form (the message list) is brought up. If we were connected, we'd tell the server we're switching there, so that the server could override the default--for example, to open the last message we were looking at, to make the app look persistent. The user may see mail messages, or they may see any empty listview, depending on how big the cache was and how recently they looked at this list. This actually happens while connected too; the only difference is that when we ask for the data then, we get it a lot faster. (There will be a visual indicator in each control that indicates the data isn't available yet; as well as a visual indicator in a status area that indicates the connection is down, so the data isn't likely to become available soon). The user could then hit the "compose" button, and the cached event for that button says to bring up the Compose Mail form. This is displayed, and the user can fill in all the controls. When finished, they hit the "Send" button. The cached event for that button says to bundle all the data from the controls, and place them in the send queue. There's still some problems we haven't worked out in the disconnected case, like what happens if the user switches apps and then comes back. While connected the server saves and restores state, but the client's not smart enough to do that on its own. TBD.

After re-establishing a connection, a few things happen. The server knows that more than one device can connect to it, so it has to pull in the saved state from the last time this particular client connected. One of the first thing the client sends is a list of items that it has thrown out of its cache (since the server keeps track of what the client has cached, and this info will be used in the next step). This info get transmitted regularly while connected; so the only odd thing here is that the client sends it all in a bundle on re-connection. The server has to note what's changed since the client was last connected, and send changes in any cached data down to the client (no need to bother with non-cached stuff). Again, while connected this gets transmitted all the time, so the only difference is that the server tries to bundle it up all at once early in the connection process. The client has to send any data that the user had entered or modified; and in fact this really is no different than data sent while connected. The data always enters a send queue and gets sent as time permits; there is just a delay when the client is disconnected.

4. when a listview is cached, does the cached data include the contents of the listed objects (for example, the contents of email messages)? Or, are the contents maintained at the server until the client requests the contents (for example, in response to the user double-clicking on a listview item?

As a general rule, the cached data does not include the contents of the listed objects. So if you go offline, you can't "open" items that you haven't opened before, even though you can see them listed. This is so we can maintain a fast response time and work on a limited memory system; we hope to have a reasonable connection. However, on systems that are reasonably fast and have lots of memory, we can store more stuff. Once you open an item, that item's data will be in the cache for a while. And in fact we will structure our apps to attach the item's data to the item's entry in the listview cache (the cache will be a table; each row represents an object, and each column a property. The list shows a couple of the properties; the object dialog shows all of them. This table starts sparse, but could at times get completely full). And in fact we hope to have a feature that allows the user to specify if they want to take the time/space hit to download the item data into the table. That way, if they're on a fat device, they can use our system pretty well even when offline--if they just visit the list, all the objects will be in the cache (assuming they have a reasonable number of objects).

5. explain any features or functions (such as local editing) that can be performed independently by the client versus features or functions that require server involvement. Explain the corresponding processing steps and interaction between client and server.

[redacted] As a simple concept to start from, just imagine that all UI is done on the client; and the server only gets involved when you have to retrieve or store/send data. Of course, it's a whole bunch more complicated than that. First, the client UI is done based on forms and events that are specified by the server. So the server has to specify them (based on the clients description of its capabilities), and send them to the client. Once

they are there, the client can do the work they specify while disconnected--but only as long as the form and events are in the cache (though they're the last thing we throw away, so hopefully they're nearly always around). Proper disconnected operation depends on the forms and events being described properly by the application; if the application does not plan ahead, then the client will have to talk to the server much more often to see what to do next. Plus, even the data can be cached. So if all forms, events, and data are cached, the user can do a whole bunch of work (well, at least view a whole bunch of data) without the server doing anything at all. Although in fact we will keep in touch with the server, letting it know what the client is doing, so the server can optionally override default actions and notify the client of real-time changes.

Another way of looking at it is that the server is in charge of everything that happens. It tells the client, in great detail, a huge number of different things that the client can do. In this sense, all work is done with server involvement. However, the client can cache anything the server sends, so it doesn't always have to talk directly to the server in order to do the same task over and over. It can process the events locally as long as they've already been done recently enough. That "recently enough" is the key to whether the server needs to be involved.

As an example, consider the scenario of adding a new contact, which could be done while connected or not. First, the user drops the app list (which is always in the client cache) and selects Contacts. If connected, the server is notified; but since it's probably been run recently, we have the Contact List form stored and marked as the default form for this app so we can bring it up even if disconnected. The form tells the client what controls to create and where to put them. The client then needs to fill the controls with the contact list. If the list is in the cache, the cached list is used, and the user is able to scroll through the list locally (unless only part of the list is cached; then as the user gets near the cache boundary, the client must ask the server for more data); otherwise, we ask the server for the list (it enters the cache quickly if connected; but could be a long wait if disconnected). Either way, the user can press the Add New button. The default event script for this says to bring up the New Contact form. Again, it should be in the cache, and is displayed. Once up, the user is free to manipulate all the controls locally, including typing information into many of them. When the Save button is pressed, all the data is bundled up and placed into the send queue. If connected, the server immediately gets it and saves the contact; otherwise, the data will just wait in the send queue until the next connection.

[redacted]

7. explain client and server actions and processing steps related to form definitions, form requests, form retrieval, form generation, form editing, etc.

The client starts by describing its capabilities--including screen size and control list--to the server. It does this when it connects for the first time.

The server--that is, an application running on the server--then figures out how a specific "form" should look. Let's say it's the Day View form for the Calendar. It specifies a menu control, some static controls, some single-line edit controls,

and maybe some buttons. It says exactly where to place each control. It also attaches some event scripts for the client to process if any of these controls are "activated" (pressing a button, selecting a menu item, tapping on a listview item are "activation" events). For some controls, it gives some naming information, so that the client knows where to look in the cache to find the data to display in the control. The Calendar app calls a server function to create the form definition; and so it's stored on the server.

At some point, the client it told to switch to that form. In this case, maybe the user selected the Calendar application for the first time. The event script attached to the "Calendar" menu item said to switch to the Day View form. The client doesn't have it, so it requests it from the server.

The server looks up the Day View name in its table of forms, and sends the definition to the client.

The client stores the Day View form in its lowest-level cache. From then on, any time the user switches to that form, it can look at the form in the cache, and it knows what controls to create, where to put them, where to find the data to put in them, and what to do when one of the controls is activated.

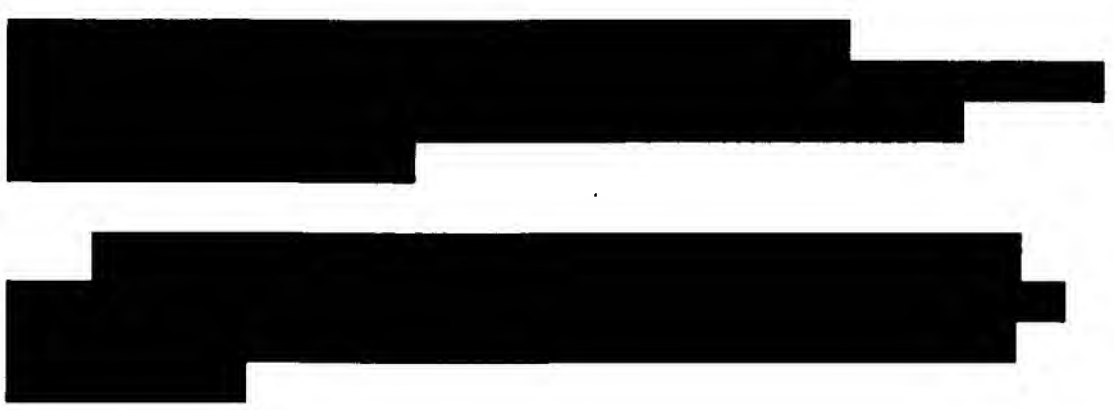
The cache entries can have date and/or version stamps on them, and the server keeps track of what it has sent to the client, so the server knows what's cached on the client. If the application is updated on the server, it could create a new form definition. The server would detect that the client version is out of date, and immediately (or on next connection) would send the new form definition to the client.

Of course, if memory got desperately tight on the client, it could always throw away the Day View form, and the cycle would start over again the next time the user chose it.

8. explain client and server actions and processing steps related to controls.

Mostly part of forms processing. The forms say what controls to create, and where. Although there can be some event scripts that say to add, remove, move, or resize controls (for example, expanding the header in a mail message). Or perhaps it could cause the naming information to change for a control, so it would get its data from a different part of the cache. The user can of course manipulate controls all they want on the client side; and when "done" with a form (usually they have to press a special button) the client follows event script that says to bundle all user-entered data from the controls and place them in the send queue for the server to process.

Although many controls have default event script attached to them that say what to do when the control is activated (which is when the user tells the control to do something--presses a button, selects a menu items, scrolls a scrollbar, etc), the event script can (and often will) include a command that says to tell the server that the control has been activated. This way, the server knows exactly what the user is doing, and so can override default event script actions.



There's really only one cache on the client. All of these objects are stored in it. However, the objects are typed, so the client knows how to act on them; and some types of objects are stored at different "levels", so they are more or less likely to be discarded. (At least in our current implementation. It's possible on some client platform that we'd change this, though probably not likely).

The separate caches are more conceptual, just so we can explain how we update different things from the server. More related to features of product.

By "control object" we mean a description of a non-standard control of some sort. This is either a control that we have added to the client platform; or a control added after our initial implementation--we can send a description of it to the client, so the client can use it, even though the client was written before we knew about the control. Can't give you too much more detail since we're not spending many cycles on this one--I kind of doubt it will be in our first product.

The "UI layout cache" is basically where all the forms are stored. The form information says how the form should look--what controls are on it, and where they are placed.

The "icon and control data cache" is where data is stored. The control descriptions inside a form specify where in the cache to find data to display inside the control.

The "event cache" is really part of the UI layout cache. It's the event script that's executed when a particular control is "activated". Although I suppose some day we could make this separate, so that you could update the events without updating the whole form.

[redacted]

[redacted]

f. explain the three client threads (command; UI; and response), especially in view of your block diagram.

On some platforms, they won't actually be separate threads. But where we can, we want them separate because we have three separate tasks to perform, and we want to keep them separate both to keep them simple and to keep the UI responsive even though communications may be occurring.

The UI thread displays things to the user, and handles user input. It reads stuff from the cache to display; if the desired item isn't in the cache, it puts a request in a queue for the command (or "send") thread. Any time the user activates a control, the UI thread immediately responds by either updating the control (if they were, say, scrolling in a listview) or by executing the event script attached to the control (if they, say, pressed a button). The event script may cause them to switch forms (which are put up by the UI thread), or send something to the server (which is placed in a queue for the command thread).

The command (or "send") thread handles data being sent to the server. It basically sits in a loop reading the send queue. If something's there, it sends it. It keeps track of what's been acknowledged; if a sent item isn't acknowledged after a period of time, it resends it. Once acknowledged, it deletes it. It calls the COMM DLL to encrypt and compress everything (so that code executes in this thread's context), and it handles reconnecting if necessary.

The response (or "receive") thread handles data coming in from the server. It is awakened whenever data comes in. The COMM DLL decrypts and decompresses it. This thread writes data into the cache, and sends a signal to the UI thread to indicate that the cache has been updated (so the UI thread might update its display, if it's related to the new cache info).

[redacted]

--Chad

From: "Chad Schwitters"
Sent: Monday, October 23, 2000 4:46 PM
To: "Peter Mansour"
Subject: caching

[redacted] If there's any details you need more of, let me know.

Thanks,
Chad

Caching--any time the server sends something to the client, the client could be caching it. In fact, it always does--the client's receive thread writes all information to the cache, and then signals the UI thread to display it. Ditto on the send side--the UI thread indicates when a send should take place, and the Send thread reads the data to be sent from the cache. (This wasn't in the original design, but Troy wanted to do it that way). The server will have some control over this, to wit:

Server side

Whenever the server sends something (form definition, icon, or form instance data) to the client, it will add it to a list it keeps. Other than some housecleaning time (must define), it will only remove items from this list when the client says it's discarding them. So when connected, this list should be an accurate representation of what the client has cached. (When not connected, the server will not do anything anyway until the client re-connects). The server can use this list to decide if server-side data changes (new mail; calendar update by secretary, etc) need to be communicated to the client. (This is more useful than just knowing the current app; since we may want to update the mail list when the user is in Contacts-at least as long as the mail list is in the cache).

Whenever the server sends any data, it will include a caching hint. This could be: don't cache this puppy at all (request a new copy every time); cache temporarily; or cache as long as you can. It can also include an optional expiration timestamp - the client is free to discard the data or form after this time (for example, a reminder for an appointment). And there can also be a timestamp or version number (which might only need to be kept on the server side; but safest to have it on both in case the server dies), so it knows when to force-feed the client a new version even though the client has it cached (for example, a form description that the client normally always has, but on connection the server can see if there's a new one installed on the server that it needs to send to the client).

Client side

The client will keep 5 lists of cached items. In the order we search them for items to toss: regular form instance data, regular icon data, form instance data hinted to keep, icon data hinted to keep, and form definitions. All 5 of these lists will be kept in most recently used last (MRUL) order. And when looking in any list, we will of course stop looking as soon as we hit any data that belongs to the current form. When we discard anything, we notify the server since it's keeping track of what we have (above). The server can also hint that something shouldn't be stuck in the cache at all.

Note that if the client goes out of range, and switches to a cached form, the data could be out of date. The user's indication of this event will be the normal UI indicator that we can't get to the server.